

Microarquitectura - Sistema de memoria

④ Jerarquía de memorias

Memoria ideal: cero latencia, cero costo, capacidad infinita

Memoria real: cuanto más capacidad tiene, más lenta es.
cuanto más rápida es, más cara.

Queremos que la memoria sea grande y rápida, para eso se diseña una jerarquía de almacenamiento:



⑤ Principios de localidad

LOCALIDAD TEMPORAL: una dirección de memoria que está siendo accedida actualmente tiene muy alta probabilidad de seguir siendo accedida en el futuro inmediato.

LOCALIDAD ESPACIAL: si se está accediendo a una dirección determinada de memoria actualmente, la probabilidad de que sus direcciones vecinas sean accedidas en el futuro inmediato es muy alta.

Los datos accedidos recientemente se guardan en memoria más rápida (cache), previendo que va a ser accedido nuevamente pronto.

Además también se guardan datos de direcciones adyacentes a la recientemente usada, para esto se divide la memoria en bloques y al necesitar un dato se guarda en cache el bloque entero al cual pertenece.

⑥ Tecnologías de memorias

MEMORIA NO VOLÁTIL: Pueden mantener la información almacenada cuando se les desconecta la alimentación.

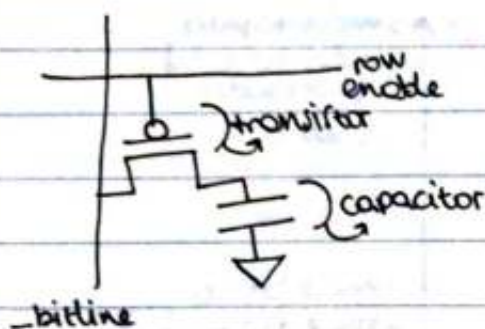
evolución:

ROM no modificables → ROM programables sólo una vez → memorias / discos de estado sólido flash

MEMORIA VOLÁTIL: Son los que se usan para implementar las memorias RAM. Cuando se interrumpe la alimentación se pierde su información. Son más rápidos que las no volátiles. Se clasifican en dinámicas (DRAM) y estáticas (SRAM).

⊗ Memoria dinámica (DRAM)

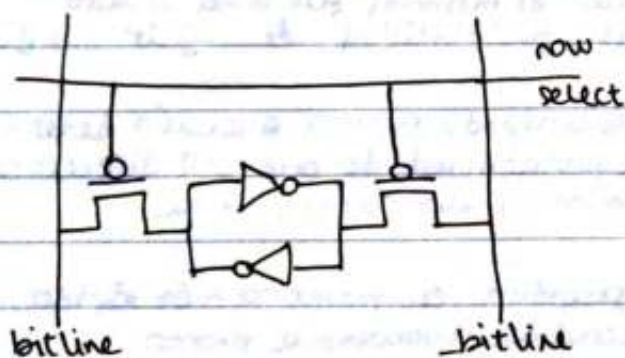
- Almacena información como el estado de carga de un capacitor
cargado $\rightarrow 1$
descargado $\rightarrow 0$
- Se accede a la celda por medio de un transistor.
- Va perdiendo energía aunque no sea leído. Tiene que ser recargado.



- Usa un transistor por unidad de memoria.
- en general está en estado de corte, consume poca energía.

- Al leer el bit se ~~descarga~~ descarga la capacidad (lectura destructiva) \rightarrow hay que regenerar la carga, por retroalimentación mediante buffers. Esto aumenta el tiempo de acceso.

⊗ Memoria estática (SRAM)



Para guardar un solo bit usa :

- 4 transistores de almacenamiento (2 en cada Δ)
- 2 transistores para acceso.

⊗ Mayor consumo por celda

⊗ Más transistores por bits

⊗ La info persiste \rightarrow no hay que recargarlo \rightarrow menor tiempo de acceso. (no destructiva).

Uso en una computadora

- = Memoria no volátil : se usa para almacenar el programa de arranque del sistema
- = El resto son memorias DRAM y SRAM : SRAM es más rápida y cara, se usa para los cachés. DRAM para la memoria principal.

MEMORIA VS. VELOCIDAD DEL PROCESADOR

- Los procesadores tienen velocidades muy superiores a las de lectura de memoria. Esto produce que cuando se necesita acceder a memoria, el procesador se quede esperando. Esto genera un cuello de botella p/ la arquitectura de Von Neumann.

Comparación DRAM / SRAM

DRAM		SRAM
consumo mínimo	✓	X Alto consumo relativo
Capacidad de almacenamiento alta (por transistor)	✓	X Muchos transistores por bit
costo bajo por bit	✓	X costo alto por bit (6 trans.)
Tiempo de acceso alto por lectura destructiva	X	✓ Tiempo de acceso rápido.

Se usa una jerarquía de memorias variando tecnología y conseguir una buena relación performance/costo.

⊗ Principio de funcionamiento de la memoria caché:

= Es un banco de SRAM de alta velocidad, que contiene una copia de datos e instrucciones de la memoria principal.

= La idea es lograr mejores tiempos de acceso haciendo que la información necesaria esté en el caché en lugar de tener que acceder a la memoria principal (DRAM más grande).

= De esta forma se busca que el programador perciba una memoria uniforme de alta velocidad pero gran capacidad.

= Requiere hardware adicional que implemente los métodos para lograr que la información/datos necesarios esté disponible. Para ello usa los principios de vecindad espacial y temporal.

⊗ Características y métricas del caché:

- TAMAÑO:
- Suficientemente grande para ser útil y proveer los datos necesarios al procesador rápidamente (el tamaño también incide en la latencia).
 - Suficientemente chica para no afectar el consumo/costo del sistema.

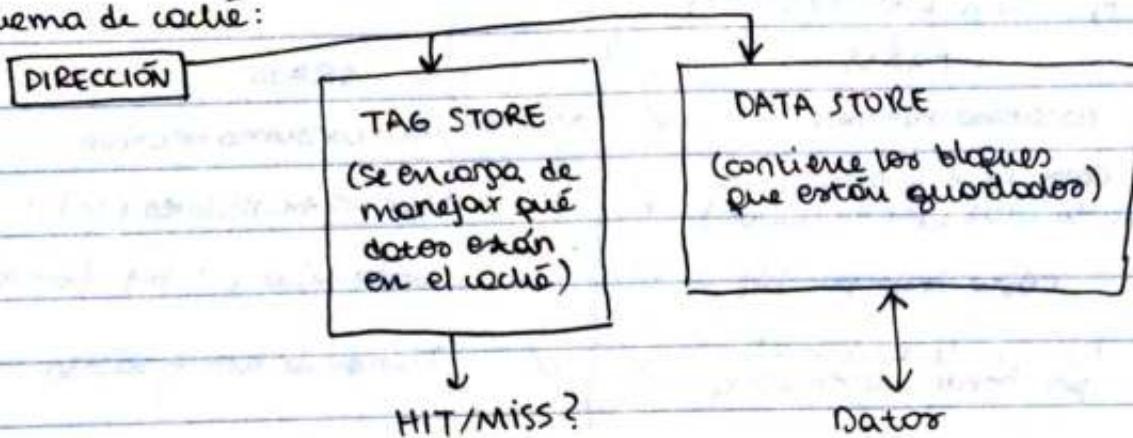
BLOQUE O LÍNEA: Unidad de almacenamiento del caché. La memoria se divide en bloques que mapean a ubicaciones en el caché.

HIT: cuando el procesador accede a un dato y este se encuentra en el caché.

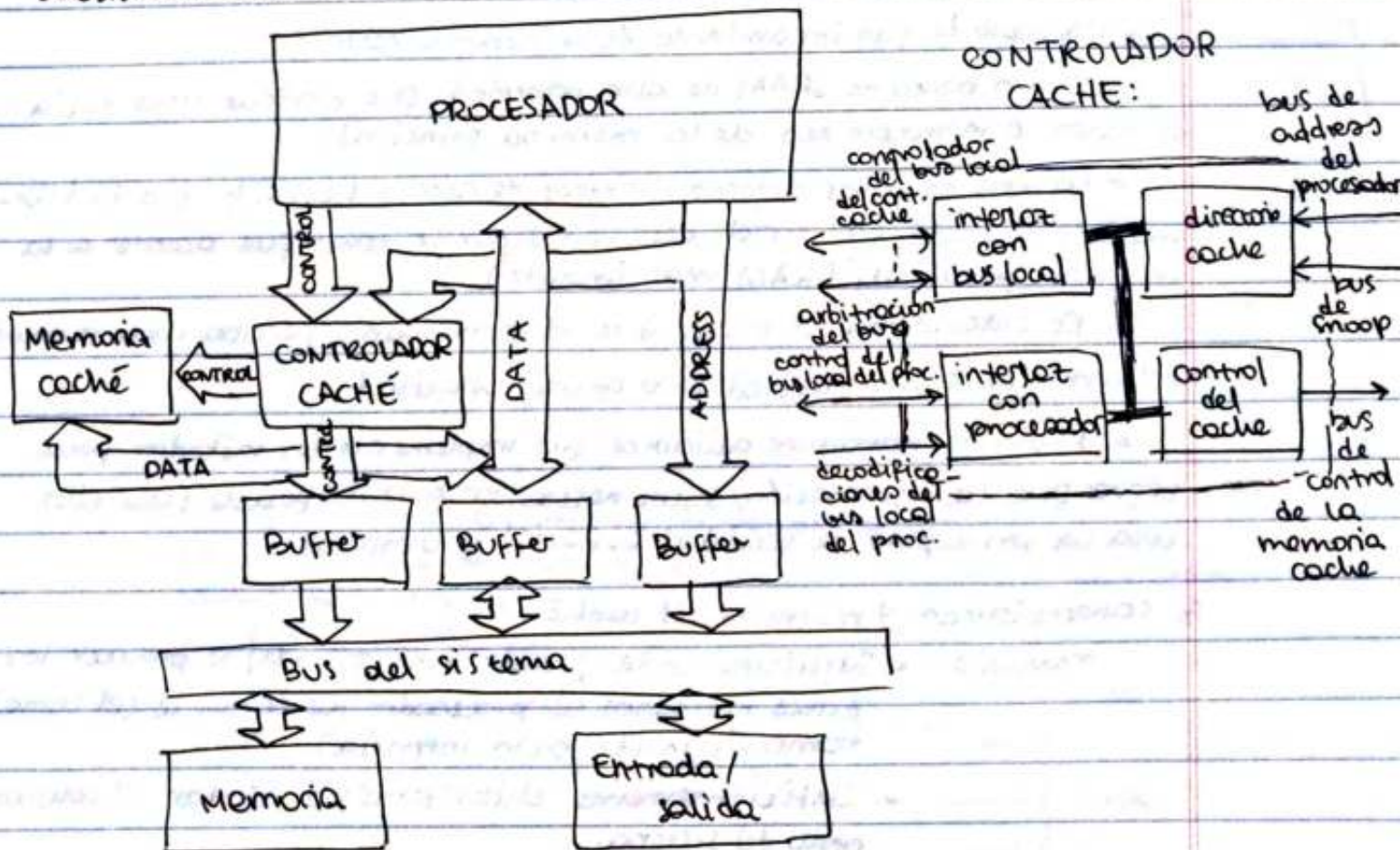
MISS: cuando el dato no se encuentra en el caché.

$$\text{HIT RATE} = \frac{\text{Accesos con hit}}{\text{Accesos totales}}$$

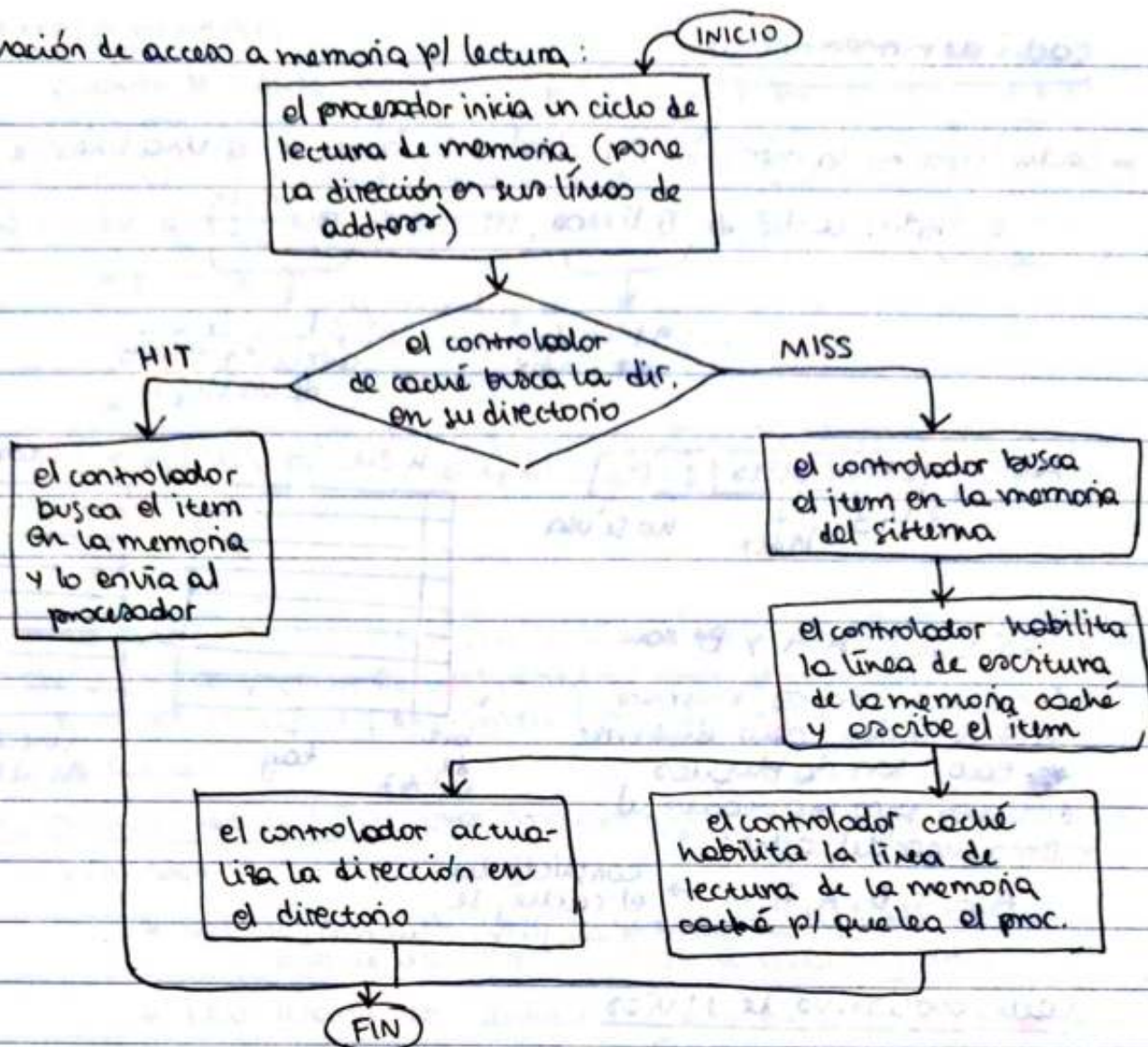
Esquema de caché:



Subsistema caché de hardware:



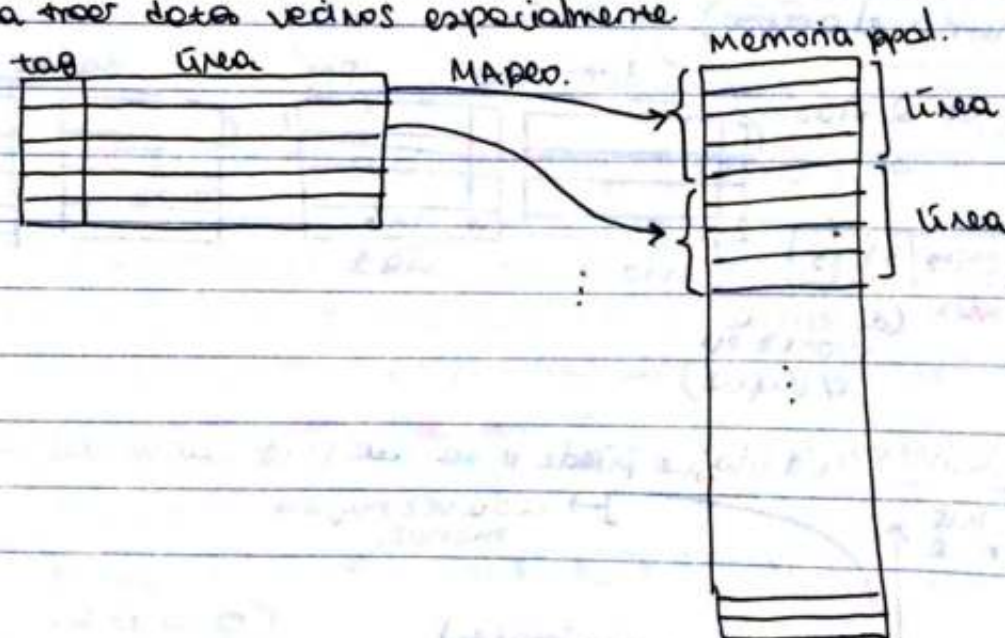
③ Operación de acceso a memoria p/ lectura:



④ Organización del caché.

LÍNEA: elemento mínimo de palabra de datos en el caché.

Es un múltiplo del tamaño de la palabra de datos en memoria, para traer datos vecinos espacialmente.



cache de mapeo directo

= cada línea en la memoria principal sólo puede ir a una línea de caché.

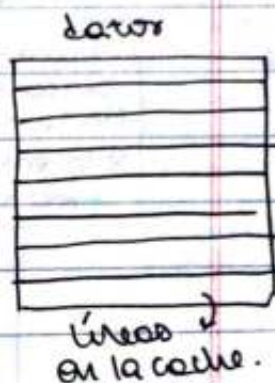
Ejemplo: caché de B líneas, bloque de B ~~words~~ ^{words}.

3 bits de ~~dirección~~ index

3 bits de dirección dentro de un bloque.

Addr:

2 bits	3 bits	3 bits
tag	index	no se usa



Problemas: si A y B son direcciones con los mismos bits de index pero diferente tag, son de bloques distintos pero mapeados al mismo lugar del caché:

A, B, A, B, A, B, ... → conflicto en el caché, se desalojan mutuamente.

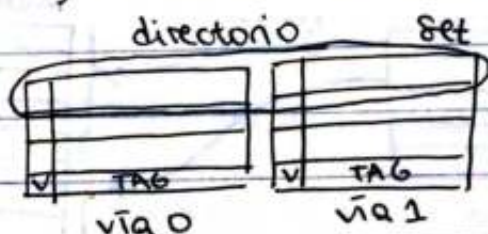
cache asociativo de N vías

= cada línea en la memoria ppal. se mapea a un SET que consiste de varias vías. De esta forma hay varios lugares posibles a los que puede ir un bloque de memoria, para reducir conflictos (pero agrega lógica, hace más lento el acceso).

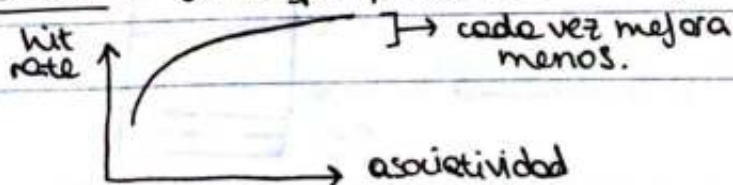
Ejemplo de 2 vías:

Addr:

3 bits	2 bits	3 bits
tag	index	(dirección words en c/bloque)



Full Associativity: un bloque puede ir en cualquier lugar del caché.



Políticas de reemplazo:

cuando se realiza una lectura para la cual hay un MISS en el caché, se busca guardar el nuevo bloque traído de memoria. Si todos los entradas a los que se mapea el nuevo bloque están llenas hay que decidir qué bloque desalojar.

(con bloques válidos)

- RANDOM

= FIFO

= LRU

- not MRU

= híbridos

= etc.

agregan lógica más o menos complicada o costosa.

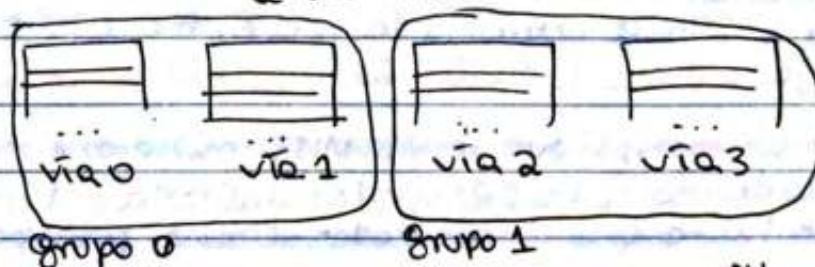
Se busca lograr una política de reemplazo eficiente al menor costo que se pueda.

- Política random: necesita lógica p/ seleccionar un bloque aleatorio del set.
- Políticas FIFO y Least Recently Used: usan alguna codificación para llevar la cuenta del orden (como en un stack) de los bloques en el caché.

Aproximaciones a LRU: usando menos lógica/bit logran resultados similares.

= Not Most Recently Used: guarda la info del último bloque usado y desaloja a cualquier otro

= LRU jerárquico: dividir los sets en grupos de acuerdo a los vías:



y para el set llevar la cuenta del grupo MRU y la vía indexada en el grupo.

- victim/nextvictim: trackea sólo el status de dos bloques victim (V) y nextvictim (NV)

todos los demás bloques son tratados como ordinarios (0).

~~se desaloja a la víctima~~
• cache MISS: se desaloja a V. NV \rightarrow V y se elige aleatoriamente un nuevo NV.

• cache hit: sólo es relevante si es hit a V o a NV (se los convierte a 0).

⊗ Escrituras en presencia de cachés:

- Una variable/dato en caché corresponde a algún lugar de la DRAM. Idealmente ambas copias deben mantener el mismo valor.
- Cuando el procesador modifica datos que están en caché se debe decidir dónde modificar la información, es decir la POLÍTICA DE ESCRITURA.
- Todo esto es más complejo si hay varios procesadores con varios cachés del mismo nivel (problema de coherencia).

Políticas de Escritura

⊗ WRITE THROUGH:

el procesador escribe a la DRAM y el controlador caché actualiza el dato en caché. Garantiza la coherencia pero penaliza los escrituras con el tiempo de acceso a la memoria principal.

⊗ WRITE THROUGH BUFFERED:

es una mejora al write through; en la que el procesador actualiza la SRAM caché y el controlador caché luego actualiza la copia en memoria DRAM mientras el procesador sigue haciendo otras cosas y accediendo al caché. El controlador debe disponer de un buffer en el cual encolar los escrituras pendientes.

⊗ COPY BACK:

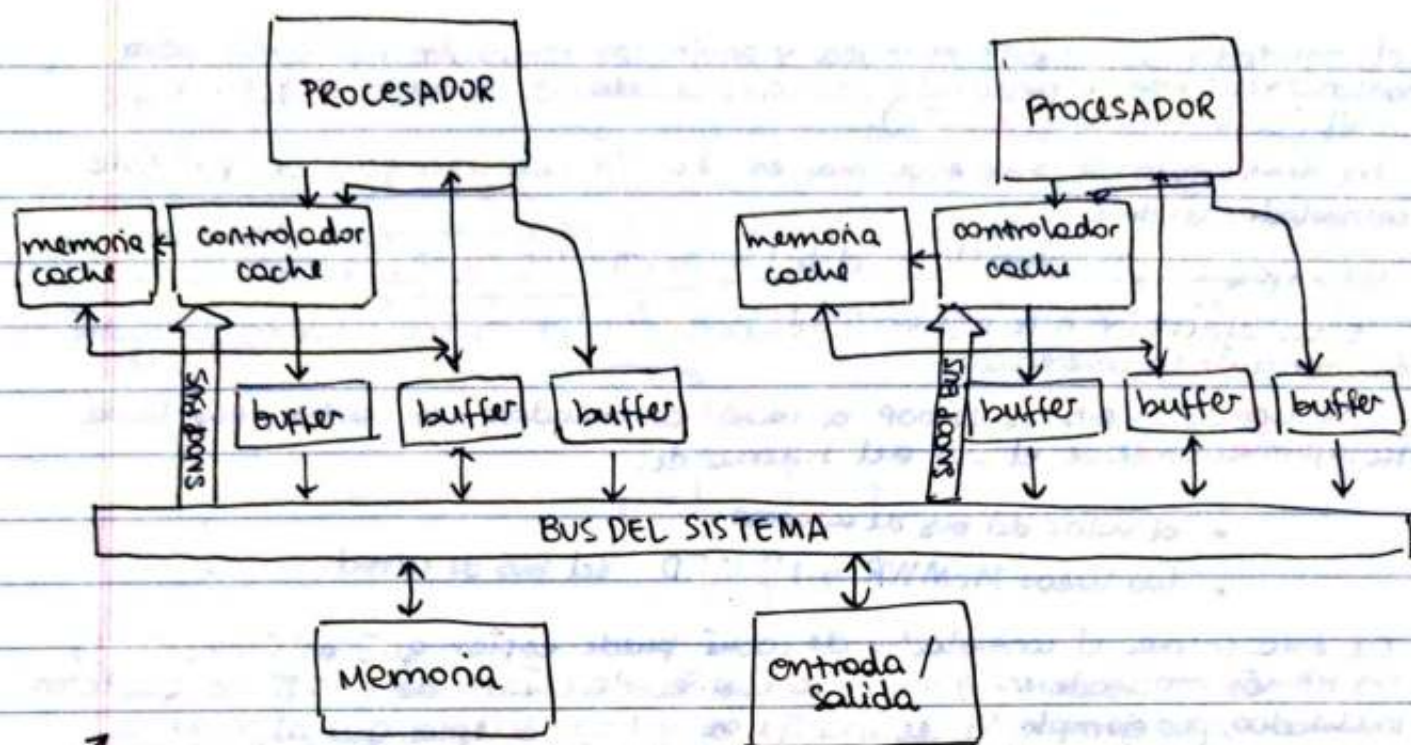
Se marcan como DIRTY los líneas de caché modificados por la escritura. Sólo se actualiza la copia en memoria DRAM cuando se desaloja la línea de la caché.

En todos los casos, si sucede un MISS mientras el controlador de caché está accediendo a la DRAM p/actualizarla, el procesador deberá esperar hasta que el controlador termine para poder acceder a la DRAM.

Coherencia en sistemas multiprocesador

cuando hay varios procesadores con su propio caché que modifican variables, los demás núcleos deben enterarse, por si tiene datos en su caché que se volvieron inválidos.

SNOOP BUS: hardware adicional para informar acerca de la escritura a caché a otros procesadores.



⊗ Esto se da en sistemas SMP (Symmetric Multi Processors)

Es un sistema con las siguientes características:

- 2 o más procesadores de capacidad similar
- estos comparten la misma memoria ppal. y dispositivos de e/s, y están interconectados por un bus teniendo aprox. el mismo tiempo de acceso a memoria.
- todos pueden realizar las mismas operaciones.
- hay un SO que provee interacción entre los procesadores y los tareas.

La organización más común es la de un bus compartido, el cual limita la velocidad de todo el sistema. Para mejorar performance, cada procesador cuenta con cachés internos. Esto introduce los problemas de coherencia entre cachés.

Si la política utilizada por todos los cachés es WRITE THROUGH es dado que no hay problemas de coherencia (pero es muy poco performante).

SOLUCIONES POR HARDWARE: Hay dos tipos de solución

PROTODOL DE DIRECTORIO y PROTODOL SNOOP

consisten de un controlador central (parte del controlador de memoria) que administra un directorio en memoria que lleva la info. de los contenidos de todos los cachés. cuando un controlador de caché realiza un pedido,

el controlador centralizado chequea y envía los comandos necesarios para conservar la coherencia (pudiendo escribir / leer datos de memoria ppal. o transferir líneas entre cachés).

La desventaja de este esquema es el cuello de botella generado por este controlador central.

Un enfoque más distribuido es el de los protocolos snoop:

estos distribuyen la responsabilidad entre todos los controladores de caché de mantener la coherencia.

Se agrega un bus de snoop a cada controladora de caché, que lleva la información desde el bus del sistema de:

- el valor del bus de address
- los líneas MEMWR y MEMRD del bus de control.

De esta forma el controlador de caché puede espiar qué están haciendo los demás procesadores y llevar la cuenta de cuáles de sus líneas quedaron invalidadas, por ejemplo. Si lee una línea y luego detecta que alguien la escribió, invalida esa línea en su caché (por ejemplo).

HAY dos enfoques → **WRITE INVALIDATE**: la idea es marcar los líneas modificados por otros como inválidos.

→ **WRITE UPDATE**: la idea es que el que modifica una línea se encarga de distribuir los datos nuevos a las cachés que la comparten.

El PROTOCOLO MESI es del tipo write/invalidate.

Cada tag de la caché tiene dos bits adicionales para codificar su estado; de entre 4 posibles:

(M) **Modified**: la línea de caché ha sido modificada, sólo está disponible en esta caché y es \neq a la info. de la memoria ppal.

(E) **Exclusive**: la línea es = a la de mem. ppal. pero no está presente en otros cachés.

(S) **Shared**: la línea es = a la de la memoria ppal. y PUEDE estar disponible en otros cachés.

(I) **Invalid**: la línea contiene datos inválidos.

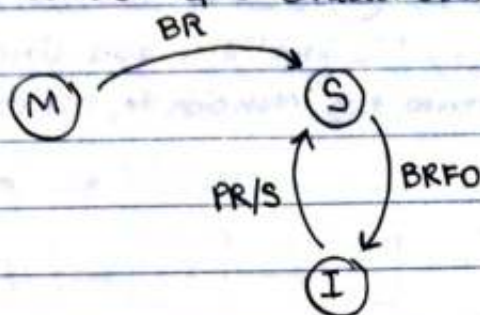
Además hay dos señales de comunicación entre cachés:

SHARED: Se activa para hacer saber a los demás que esa línea es válida en su caché (y que no es exclusiva).

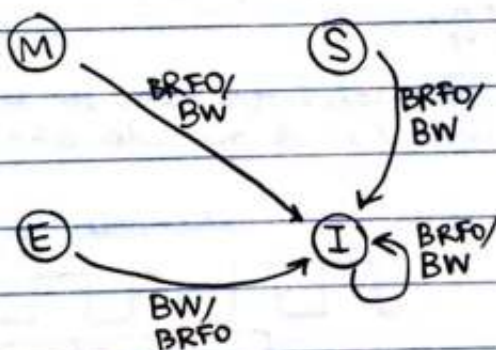
RFO (Request For Ownership) se activa para pedir exclusividad de una línea (y poder modificarla).

Uso de RFO

⊗ Cuando un caché tiene una línea en estado M debe hacer Snoop en el bus, y si detecta una señal de lectura por parte de otro procesador (BR) lo que hace es emitir la señal de RFO para avisarle que los datos en memoria ppal. son inconsistentes, escribir a memoria y enviarte los datos al lector. Para a estar en estado shared (S) ambos, y los cachés que tenían esa línea la invalidan al ver el RFO:



⊗ cuando una caché tiene una línea en estado S o I y quiere escribir a ella, los demás deben invalidar esa línea: para eso envía el RFO.



⊗ los estados M y E son privados: esta línea sólo se encuentra en esta caché. En cambio una línea en estado S podría ser exclusiva sin saberlo.

- Una escritura sólo puede realizarse libremente sobre una línea en estado M o E.

escritura

- En el caso de una línea compartida S primero deben invalidarse las otras copias enviando la señal RFO. (lo mismo con I probablemente).

- Un caché que tiene una línea en estado M debe hacer snoop en el bus para detectar lecturas a esa línea.

líneas no modificadas.

Esto se logra haciendo que la lectura haga retry un tiempo después y escribiendo la info a memoria ppal. (M → S)

= o bien enviando la línea al caché que le quiere leer directamente (y escribiendo a memoria x consistencia de los estados)

→ toda lectura a una línea que está M o E en otro caché produce un cache MISS (ya que son estados exclusivos).

Un caché con una línea (S) debe escuchar los broadcasts RFO de otras cachés para invalidarla en ese caso ((S) → (I))

READ FOR OWNERSHIP: combina una lectura con un RFO (broadcast de invalidación). ~~Se usa cuando se trata de escribir~~ Se usa cuando se trata de escribir a una línea en estado (S) o (I).

causa que los demás cachés invaliden esa línea.
Es una operación de lectura con intención de escritura.

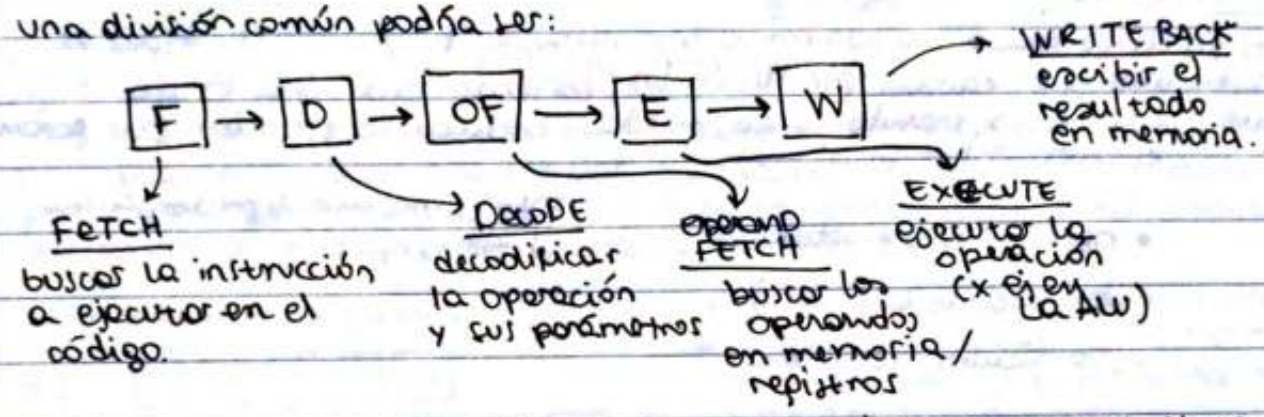
Microarquitectura: Paralelismo a Nivel de Instrucción (ILP)

PIPELINING

El objetivo de esta técnica es aumentar el throughput de instrucciones (instrucciones ejecutadas por unidad de tiempo). Consiste en dividir el ciclo de instrucción en varias etapas, con el objetivo de paralelizarlos y ejecutar etapas distintas de instrucciones distintas al mismo tiempo, aprovechando al máximo el hardware disponible.

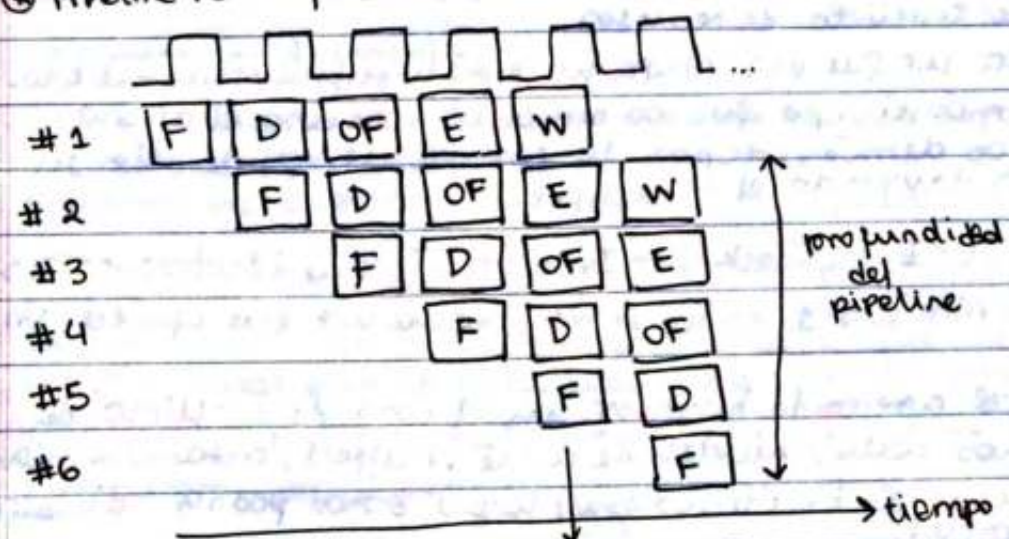
La latencia de cpu de las instrucciones no disminuye, e incluso podría tener un poco de overhead a causa de la división en etapas.

una división común podría ser:



Puede verse que distintas instrucciones usan algunas etapas y no todas. Esto podría alargar la latencia de estas operaciones en particular.

⊗ Pipeline ideal: paraleliza todo.



8:30 - 10 (238) 2do piso
10 - 11:30 (62) Subnivel planta 46-47

condición de régimen (pipeline lleno) } luego de N ciclos de clock.

luego de que el pipeline llega a la condición de régimen, idealmente completa una instrucción por ciclo de reloj.

Aumentar la cantidad de etapas (haciéndolos más cortos) hasta que se completaron más instrucciones por unidad de tiempo.

Esto es así porque

$$\text{TPI} = \frac{\text{latencia de una instrucción (máxima)}}{\text{etapas del pipeline}}$$

tiempo que tarda
en ejecutar una instrucción
en promedio en régimen de
pipeline (throughput)

OBSTÁCULOS

Los pipelines reales no son como los ideales: hay distintos tipos de obstáculos que causan pipeline stalls, es decir que una etapa tenga que quedarse esperando y no pueda continuar la ejecución que formaría el throughput ideal:

- Obstáculos estructurales
- Obstáculos de control
- Obstáculos de datos.

Esto causa una degradación en el throughput.

① Obstáculos estructurales

Suceden cuando el hardware no soporta la combinación de etapas entre dos instrucciones. Por ejemplo un Fetch y un Operand Fetch concurrentes, cualquier tipo de conflicto de recursos.

También podría ser que una etapa no esté lo suficientemente atomizada y tome más tiempo que los demás. En ese caso el diseño haría que en los demás etapas se gaste tiempo de reloj sin hacer nada, disminuyendo el throughput.

Esto afecta el CPI (Clock Per Instruction) que idealmente es 1, aumentándolo en al menos 1 ciclo de reloj cada vez que aparece un obstáculo.

Pueden resolverse agregando hardware en el caso de conflicto de recursos, como más caché / niveles de caché, buffers, ensanchamiento de buses. En el caso de ~~las~~ etapas muy largas estas podrían dividirse en etapas más simples.
(aumento de la profundidad del pipeline).

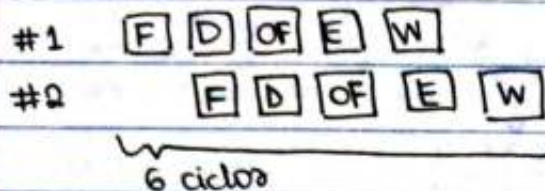
② Obstáculos de datos

Se producen cuando x efecto del pipeline, una etapa de una instrucción concurre con una anterior necesita un dato que aún no está disponible.

Por ejemplo: $\text{ADD } R1 \leftarrow R2, R3 \quad \#1$
 $\text{SUB } R4 \leftarrow R1, R2 \quad \#2$

En este caso se produce una dependencia de datos en el registro R1. Hasta que el ADD no finaliza, el resultado no está disponible para el SUB.

Pipeline Ideal:



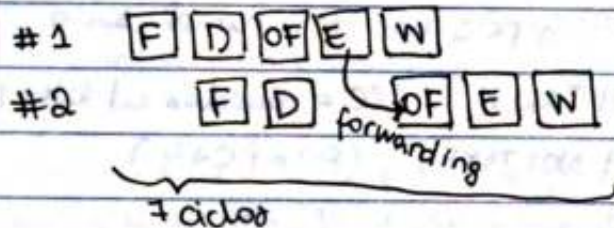
Pipeline stall:



* Solución a obstáculos de datos: Data Forwarding

Consiste en agregar el hardware necesario para llevar el resultado de la operación que hay que esperar (de la unidad de ejecución correspondiente) directamente al inicio de la etapa que está esperando, ahorrando la etapa de escritura al operando de destino.

Incidencia en el pipeline:



Hay casos para los cuales no sirve: por ejemplo si la primera instrucción trae un dato de memoria a un registro hay que esperar que termine su etapa W (se forwardea desde la AW):

$\text{LD } R1 \leftarrow [\text{0xAAAA}]$
 $\text{SUB } R3 \leftarrow R1, R2$

* Obstáculos de control

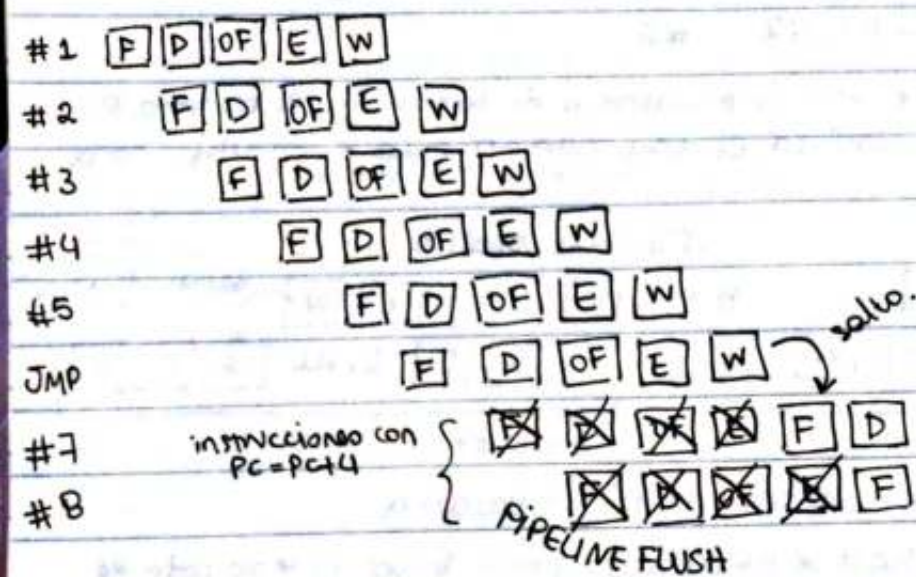
Ocurren cuando hay branches o saltos en el código. En general, cuando una operación no es de control de flujo, se sabe que la siguiente va a estar en $\text{PC} = \text{PC} + 4$ (dependiendo del tamaño).

Si consideramos que las instrucciones se van procesando secuencialmente en el pipeline, cuando se llega a un branch que produce un salto y a sea hacia adelante o hacia atrás, todo lo que se encontraba en el pipeline antes debe ser descartado. Esto se conoce como **BRANCH PENALTY** y desperdicia $N-1$ ciclos de procesador ($N = \#$ de etapas del pipeline).

ES la peor situación posible de pérdida de performance.

Ejemplo: $I_1 \ I_2 \ I_3 \ I_4 \ I_5 \ \text{JMP} \ I_7 \ I_8 \ I_9 \ I_{10} \ I_{11} \ I_{12} \dots$

no son saltos



Quando el salto es incondicional, y se puede determinar su dirección en la etapa de ~~decodif.~~, esto no representa un problema ya que se conoce el próximo PC. ^{codif.}

El peor inconveniente está en el caso de los saltos condicionales, que dependen de la ejecución/evaluación de la condición para tomar o no el salto.

CONDICIÓN = TRUE \rightarrow BRANCH TAKEN (PC = dirección del salto)

CONDICIÓN = FALSE \rightarrow BRANCH NOT TAKEN (PC = PC + 4)

⊗ Determinación de la dirección de saltos

⊗ Salto condicional: se determinan en la fase **E**

⊗ Salto incondicional: \rightarrow direccionamiento indirecto **OF**
(está en memoria o en un registro)

direccionamiento directo
(la dirección está explícita en la op.) **D**

• Se puede usar data forwarding para conocer antes los resultados de los branches, y así disminuir la branch penalty.

• Lo más efectivo es realizar predicción de saltos.